Returning to the Condorcet criterion, the problem is to find a method that will pick the Condorcet winner of an election, if such a winner exists. A Condorcet winner might not necessarily exist. For example, say we have three candidates, $A$, $B$, and $C$, for which we received the following ballots:

30: $[A, B, C]$
30: $[B, C, A]$
30: $[C, A, B]$

If we do the pairwise comparisons, then we find that $A$ beats $B$ by 60 to 40, $B$ beats $C$ by 60 to 40, and $C$ beats $A$ by 60 to 40. Therefore each candidate beats some other candidate, and no candidate beats more candidates than others, so no overall winner emerges.

Now see another election involving three candidates, where the ballots were cast in a different way:

$10 \times [A, B, C]$
$5 \times [B, C, A]$
$5 \times [C, A, B]$

$A$ is preferred over $B$ by 15 to 5, $B$ is preferred over $C$ by 15 to 5, and $C$ and $A$ tie up with 10 each. Because both $A$ and $B$ have a pairwise win, we cannot declare a winner.

That's a bit strange because it is not a case where all ballots take an equal share of the voters, as the previous one. It is clear that more voters cast the first ballot, but somehow this does not make its way to the results. We would therefore like to have a method that respects the Condorcet criterion but is less prone to ties than the simple method we have been using until now.

## 10.2 The Shulze Method

A method that finds the Condorcet winner of an election, if such a winner exists, is the Schulze method, developed in 1997 by Markus Schulze. It is a method used among technological organizations, and is not prone to ties. The basic idea in the Schulze method is that we use the pairwise preferences of the voters to construct a graph. Then the preferences between candidates are found by tracing paths on that graph.

The first step in the Schulze method is exactly finding the pairwise preferences for the candidates. Suppose we have $n$ ballots and $m$ candidates. The ballots are $B = B_1, B_2, \ldots, B_n$ and the candidates are $C = c_1, c_2, \ldots, c_m$. We take

each ballot $B_i$ in turn, where $i = 1, 2, \ldots n$. Each ballot contains a series of candidates, in decreasing preference order, so a candidate who precedes other candidates in a ballot is preferred by the voter who cast that ballot to any of the candidates who follow in the ballot. In other words, for any two candidates $c_j$ and $c_k$ in ballot $B_i$, the voter prefers $c_j$ to $c_k$ if $c_j$ appears before $c_k$. To tally the preferences, we use an array $P$ of size $m \times m$. To calculate the contents of the array, we first initialize it to all zero values. Then as we read each ballot $B_i$, for every pair $c_j$ and $c_k$ of candidates in the ballot such that $c_j$ precedes $c_k$, we add one to element $P[c_j, c_k]$ of array $P$. When we are done reading all ballots, each element of $P$, $P[c_j, c_k]$, will show how many voters in total prefer candidate $c_j$ to candidate $c_k$.

For example, a ballot might be $[c_1, c_3, c_4, c_2, c_5]$, meaning that the voter prefers candidate $c_1$ over all other candidates, candidate $c_3$ over candidates $c_4$, $c_2$, $c_5$, candidate $c_4$ over candidates $c_2$, $c_5$, and candidate $c_2$ over candidate $c_5$. In that ballot candidate $c_1$ is before candidates $c_3$, $c_4$, $c_2$, and $c_5$, so we add one to elements $P[c_1, c_3]$, $P[c_1, c_4]$, $P[c_1, c_2]$, and $P[c_1, c_5]$. Candidate $c_3$ is before candidates $c_4$, $c_2$, and $c_5$, so we add one to elements $P[c_3, c_4]$, $P[c_3, c_2]$, and $P[c_3, c_5]$, and so on, up to candidate $c_2$, where we add one to element $P[c_2, c_5]$:

$$\begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{array} \begin{array}{c} \begin{array}{ccccc} c_1 & c_2 & c_3 & c_4 & c_5 \end{array} \\ \begin{bmatrix} - & +1 & +1 & +1 & +1 \\ - & - & - & - & +1 \\ - & +1 & - & +1 & +1 \\ - & +1 & - & - & +1 \\ - & - & - & - & - \end{bmatrix} \end{array}$$

Algorithm 10.1 does that. The algorithm creates the array $P$ that will hold the pairwise preferences in line 1 and initializes the pairwise preferences for all candidates to 0 in lines 2–4. This requires $\Theta(|B|^2)$ time. Then it takes each of the $|B|$ ballots, in the loop in lines 5–11. In each ballot, it takes each candidate in turn in the nested loop in lines 7–11. Because the ballots contain the candidates in the preference of the voter, each time we enter the nested loop, the selected candidate is preferred by all the following candidates in the ballot. So, if the candidate is $c_j$, then for every other candidate $c_k$ that follows $c_j$ in the ballot, we add one to the element $P[c_j, c_k]$. It does the same thing starting with all the other candidates in the ballot in sequence. If the ballot contains all $|C|$ candidates, then it will update the preferences array $(|C| - 1) + (|C| - 2) + \ldots + 1 = |C|(|C| - 1)/2$ times. In the worst case all ballots contain all $|C|$ candidates, so the time required for each ballot is

**Algorithm 10.1**: Calculate pairwise preferences.

`CalcPairwisePreferences`$(ballots, m) \to P$

**Input**: *ballots*, an array of ballots, where each ballot is an array of candidates
$m$, the number of candidates

**Output**: $P$, an array of size $m \times m$ with the pairwise preferences for candidates; $P[i, j]$ is the number of voters that prefer candidate $i$ to candidate $j$

```
1  P ← CreateArray(m · m)
2  for i ← 0 to m do
3      for j ← 0 to m do
4          P[i][j] ← 0
5  for i ← 0 to |ballots| do
6      ballot ← ballots[i]
7      for j ← 0 to |ballot| do
8          c_j ← ballot[j]
9          for k ← j + 1 to |ballot| do
10             c_k ← ballot[k]
11             P[c_j, c_k] ← P[c_j, c_k] + 1
12 return P
```

$O(|C|(|C| - 1)/2) = O(|C|^2)$ and the time for all ballots is $O(|B||C|^2)$. In total, algorithm 10.1 runs in $O(|C|^2 + |B|^2)$ time.

As an example, take an election with four candidates, $A$, $B$, $C$, and $D$. In the election 21 voters took part. After counting the ballots we found that the following had been cast:

$$6 \times [A, C, D, B]$$
$$4 \times [B, A, D, C]$$
$$3 \times [C, D, B, A]$$
$$4 \times [D, B, A, C]$$
$$4 \times [D, C, B, A]$$

That is, six ballots $[A, C, D, B]$, four ballots $[B, A, D, C]$, and so on. In the first six ballots, the voters preferred candidate $A$ to candidate $C$, candidate $C$ to candidate $D$, and candidate $D$ to candidate $B$.

To calculate the preference array, we find that candidate $A$ is preferred to $B$ in only the first set of ballots, so the entry in the array for the pairwise preferences between $A$ and $B$ will be 6. Similarly, we find that candidate $A$ is preferred to $C$ in the first, second, and fourth set of ballots, so the pairwise preferences between $A$ and $C$ will be 14. Continuing in this way, we find that the preference array for the candidates of our election is:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 6 & (6+4+4) & (6+4) \\ (4+3+4+4) & 0 & (4+4) & 4 \\ (3+4) & (6+3+4) & 0 & (6+3) \\ (3+4+4) & (6+3+4+5) & (4+4+4) & 0 \end{bmatrix} \end{array}$$

that is:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 6 & 14 & 10 \\ 15 & 0 & 8 & 4 \\ 7 & 13 & 0 & 9 \\ 11 & 17 & 12 & 0 \end{bmatrix} \end{array}$$

The second step of the Schulze method starts by constructing a graph, where the candidates are the nodes and the margins of the preferences of one candidate over another are the weights of the links. If for two candidates $c_i$ and $c_j$ the number $P[i,j]$ of voters that prefer $c_i$ over $c_j$ is greater than the number of voters $P[j,i]$ that prefer $c_j$ over $c_i$, we add the link $c_i \to c_j$ and we assign the number $P[i,j] - P[j,i]$ as the weight of the link $c_i \to c_j$. We use $-\infty$ for the other pairs to show that the corresponding link does not exist. To do that we start by doing the comparisons and operations as necessary:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & (6<15) & (14-7) & (10<11) \\ (15-6)=9 & 0 & (8<13) & (4<17) \\ (7<14) & (13-8) & 0 & (9<12) \\ (11-10)=1 & (17-4) & (12-9) & 0 \end{bmatrix} \end{array}$$

and then substituting $-\infty$ for negative and zero entries:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} -\infty & -\infty & 7 & -\infty \\ 9 & -\infty & -\infty & -\infty \\ -\infty & 5 & -\infty & -\infty \\ 1 & 13 & 3 & -\infty \end{bmatrix} \end{array}$$

You can see the corresponding graph in figure 10.1. As explained, the graph has the candidates as its nodes and the positive margins of the differences in preferences between each pair of candidates as its edges.

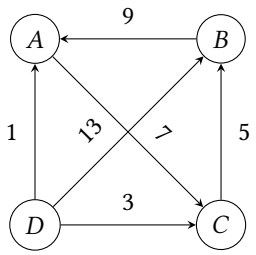



**Figure 10.1**
An election graph.

Having constructed the graph, we proceed to calculate the *strongest paths* between all nodes in the preference graph. We define the *strength of a path*, or equivalently the *width of a path*, as the minimum weight in the links that make up the path. If you imagine a path as a sequence of bridges between nodes, then the path is only as strong as its weakest link, or bridge. There may be multiple paths between two nodes in the preference graph, each with a different strength. The path with the greatest strength among them is the strongest path. Returning to the bridges metaphor, the strongest path is the path that allows us to transport the heaviest vehicle between the two nodes. Figure 10.2 shows a graph and two strongest paths. The strongest path between node 0 and node 4 passes through node 2 and has strength 5; similarly, the strongest path between node 4 and node 1 passes through node 3 and has strength 7.

Finding the strongest paths is a problem that arises in other areas as well. In computer networks, it is equivalent to finding the maximum bandwidth between two computers in the Internet when between any two computers or routers in the network the link has a limited bandwidth capacity. It is also called the *widest path problem* because strength is synonymous with width, and *maximum capacity path problem* because it boils down to finding the maximum capacity of a path in a graph. The maximum capacity of the path is constrained by its weakest link; the maximum capacity path between two nodes is the maximum capacity among the capacities of the paths between the two nodes.

To find the strongest path between all pairs of nodes in the graph, we reason as follows. We take all nodes of the graph in sequence, $c_1, c_2, \ldots, c_n$. We find

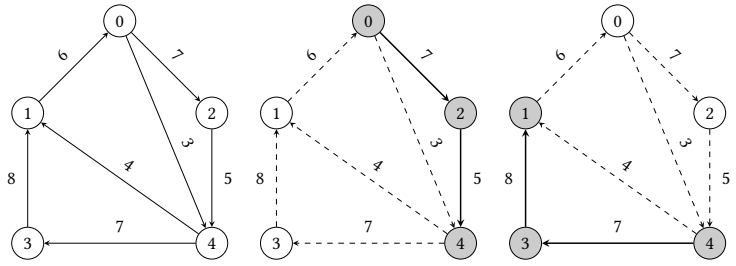



**Figure 10.2**
Strongest path examples.

the strongest path between all pairs of nodes, $c_i$ and $c_j$ in the graph using zero intermediate nodes from the sequence $c_1, c_2, \ldots, c_n$. The strongest path between the two nodes using no intermediate nodes exists if there is a direct link between $c_i$ and $c_j$; otherwise it does not exist at all.

We proceed to find again the strongest path between all pairs of nodes $c_i$ and $c_j$ using the first of the nodes in the sequence, node $c_1$, as an intermediate node. If we have already found a strongest path between $c_i$ and $c_j$ in the previous step, then if there exist paths $c_i \rightarrow c_1$ and $c_1 \rightarrow c_j$, we compare the strength of the path $c_i \rightarrow c_j$ with the strength of the path $c_i \rightarrow c_1 \rightarrow c_j$, and we take the strongest of the two as our new estimate for the strongest path between $c_i$ and $c_j$.

We continue the same procedure until we have used all $n$ nodes in the sequence. Suppose we have found the strongest path between all pairs of nodes in the graph, $c_i$ and $c_j$, using the first $k$ nodes of the sequence as intermediate nodes. We now try to find the strongest path between the two nodes $c_i$ and $c_j$ using the first $k+1$ nodes in the sequence. If we find a path from $c_i$ to $c_j$ using the $(k+1)$th node, it will consist of two parts. The first part will be a path from $c_i$ to $c_{k+1}$ using the first $k$ nodes of the sequence as intermediate nodes, and the second part will be from $c_{k+1}$ to $c_j$ using again the first $k$ nodes of the sequence as intermediate nodes. We have already found the strengths of these two paths in the previous steps of the procedure. If we call $s_{i,j}(k)$ the strength of the path from $c_i$ to $c_j$ using the first $k$ nodes, the two paths using node $(k+1)$ are $s_{i,k+1}(k)$ and $s_{k+1,j}(k)$. The strength of the path going from $c_i$ to $c_j$ through $c_{k+1}$ is by definition the minimum of the strength of the paths $s_{i,k+1}(k)$ and $s_{k+1,j}(k)$ and we have already found both of them previously. So:

$$s_{i,j}(k+1) = max\Big(s_{i,j}(k), min\big(s_{i,k+1}(k), s_{k+1,j}(k)\big)\Big)$$

In the end, after using all $n$ nodes in the sequence, we will have found all the strongest paths between any two pairs in the graph. Algorithm 10.2 shows the procedure in detail.

In lines 1–2 of the algorithm we create the two output arrays. In lines 2–10, we initialize the strength of the path between two nodes to their direct links between them, if they exist. They correspond to finding the strongest paths using zero intermediate nodes. Then in lines 11–18, we calculate strongest paths using more and more intermediate nodes. The outer loop, over variable $k$, corresponds to the intermediate node we are adding to our set of intermediate nodes. Each time we increase $k$ we check all pairs of nodes, given by $i$ and $j$, and adjust their strongest path estimates, if needed, in lines 14–16. We also keep track of the paths using an array *pred* that gives, at each position $(i, j)$ in the array, the predecessor node along the strongest path from node $i$ to node $j$.

The algorithm is efficient. The first loop in lines 2–10 executes $n^2$ times and the second loop in lines 11–18 executes $n^3$ times, where $n$ is the number of vertices in the graph. Because the graph is represented as an adjacency matrix, all graph operations take constant time, so in total it takes $\Theta(n^3)$ time. If we take the discussion to elections, then $n$ is the number of candidates. In the notation we have adopted, the time required is $\Theta(|C|^3)$.

You can see a trace of the execution of the algorithm in our example in figure 10.3. At each sub-figure we indicate with a gray fill the intermediate node that is used to form a new path at each step. Note that in the last step, when we add node $D$ in our set of intermediate nodes, we find no path stronger from those already found. This may happen, but we cannot know it in advance. It is also possible to happen in one of the previous steps, even though it did not happen in this case. Be that as it may, we have to carry out the execution of the algorithm to the end, examining all nodes of the graph in sequence as intermediate nodes.

The paths are returned from algorithm 10.2 via array *pred*. If you want to see the paths on the graph, you can check them in figure 10.4. Verify that the strongest path from node $D$ to node $C$ is through nodes $B$ and $A$ and not the direct path from $D$ to $C$; at the third sub-figure the earlier estimate gets overwritten by the better estimate when we use node $B$ as an intermediate node. By contrast, when we add $A$ as an intermediate node we can obtain the path $D \longrightarrow A \longrightarrow B$ with strength 1; but we already have path $D \longrightarrow A$ with strength 1, so there is no need to update the strongest path between $A$ and $D$. By the same token, we can obtain the path $D \longrightarrow A \longrightarrow C$ with strength 1; but

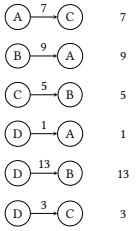
**Algorithm 10.2:** Calculate strongest paths.

`CalcStrongestPaths(`$W$, $n$`)` $\rightarrow (S, pred)$

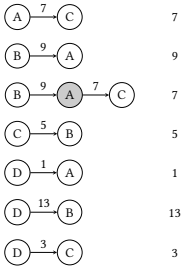
**Input**: $W$, an array of size $n \times n$ representing the adjacency matrix of a graph; $W[i, j]$ is the weight of the edge between nodes $i$ and $j$
$n$, the size of each dimension of $W$

**Output**: $S$, an array of size $n \times n$ such that $S[i, j]$ is the strongest path between nodes $i$ and $j$
$pred$, an array of size $n \times n$ such that $pred[i, j]$ is the predecessor of node $i$ in the strongest path to node $j$

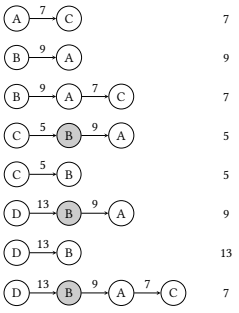
```
1  S ← CreateArray(n · n)
2  pred ← CreateArray(n · n)
3  for i ← 0 to n do
4      for j ← 0 to n do
5          if W[i, j] > W[j, i] then
6              S[i][j] ← W[i, j] − W[j, i]
7              pred[i, j] ← i
8          else
9              S[i][j] ← −∞
10             pred[i, j] ← −1
11 for k ← 0 to n do
12     for i ← 0 to n do
13         if i ≠ k then
14             for j ← 0 to n do
15                 if j ≠ i then
16                     if S[i, j] < Min(S[i, k], S[k, j]) then
17                         S[i, j] ← Min(S[i, k], S[k, j])
18                         pred[i, j] ← pred[k, j]
19 return (S, pred)
```



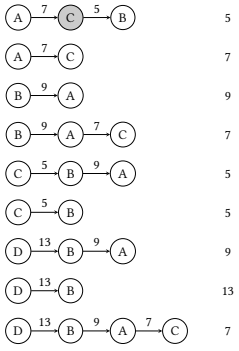

(a) No intermediate nodes.



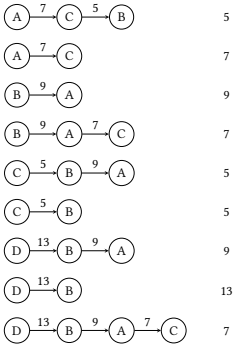

(b) $A$ as intermediate node.




(c) $A$, $B$ as intermediate nodes.




(d) $A$, $B$, $C$ as intermediate nodes.




(e) $A$, $B$, $C$, $D$ as intermediate nodes.

**Figure 10.3**
Calculation of strongest paths.

we already have path $D \to C$ with strength 3, so the strongest path between $D$ and $C$ remains unchanged.

What the algorithm really does is calculate parts of the solution to our problem and combine them incrementally to arrive at the overall solution. It finds shorter strongest paths that it combines, if possible, producing longer strongest paths. This strategy, of solving part of the problem and combining the solved parts to produce the final solution, is called *dynamic programming* and lies behind many interesting algorithms.

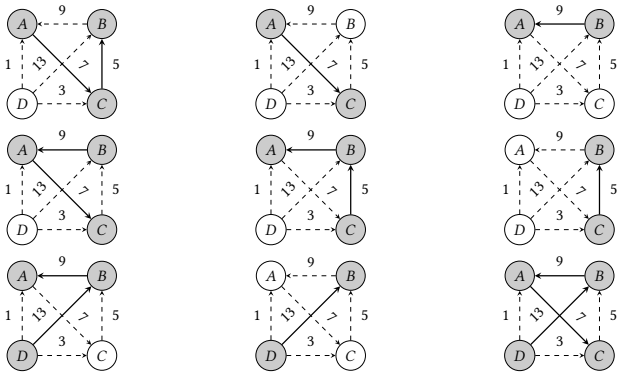



**Figure 10.4**
Strongest paths for the election graph.

In our example, algorithm 10.2 produces an array $S$ with the strengths of the strongest paths between any pair of nodes:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} -\infty & 5 & 7 & -\infty \\ 9 & -\infty & 7 & -\infty \\ 5 & 5 & -\infty & -\infty \\ 9 & 13 & 7 & -\infty \end{bmatrix} \end{array}$$

With these available, we can go on in the third step of the Schultze method. We have established, for any two candidates $c_i$ and $c_j$ how much support there is for candidate $c_i$ to $c_j$ as well as how much support there is for candidate $c_j$ to $c_i$. They are the strengths of the paths from $c_i$ to $c_j$ and from $c_j$ to $c_i$, respectively. If the strength of the path from $c_i$ to $c_j$ is greater than the strength of the path from $c_j$ to $c_i$, then we say that candidate $c_i$ wins over candidate $c_j$. We want then to find, for each candidate $c_i$, over how many other candidates

**Algorithm 10.3**: Calculate results.

```
CalcResults(S, n) → wins
    Input: S, an array of size n × n with the strongest paths between
           nodes; s[i, j] is the strongest path between nodes i and j
           n, the size of each dimension of S
    Output: wins, a list of size n; item i of wins is a list containing m
            integer items j_1, j_2, ..., j_m for which s[i, j_k] > s[j_k, i]
1   wins ← CreateList()
2   for i ← 0 to n do
3       list ← CreateList()
4       InsertInList(wins, NULL, list)
5       for j ← 0 to n do
6           if i ≠ j then
7               if S[i, j] > S[j, i] then
8                   InsertInList(list, NULL, j)
9   return wins
```

$c_i$ wins. That is easy to do, by just going over the array with the path strengths that we calculated using algorithm 10.2 and adding up the times $c_i$ is preferred over $c_j$. We do this in algorithm 10.3, which returns a list *wins* such that item $i$ of the list *wins* contains a list with the candidates over which candidate $i$ prevails.

In our example we find that $A$ beats $C$, $B$ beats $A$ and $C$, $C$ is beaten by all, and $D$ beats $A$, $B$, and $C$. In particular, $wins = [[2], [2, 0], [], [2, 1, 0]]$. Because the number of times a candidate wins other candidates is $A = 1$, $B = 2$, $C = 0$, and $D = 3$, $D$ is the preferred candidate. Algorithm 10.3 requires $O(n^2)$ time, where $n$ is the number of candidates, or $|C|$ in the notation that we have been using, so the time required is $O(|C|^2)$. Because algorithm 10.1 runs in $O(|C|^2 + |B|^2)$ time, algorithm 10.2 runs in $\Theta(|C|^3)$ time, and algorithm 10.3 runs in $O(|C|^2)$ time, the whole Schulze method requires polynomial time and is efficient.

Note that we did not just get a winner for the election; we got an ordering of the candidates. Thus, the Schulze method can also be used when we want to pick the first $k$ out of a total $n$ candidates. We just select the first $k$ in the ordering it produces.

In our example we did not have ties. Candidates were ordered in a clear way. That may not always happen. The Schulze method will produce a Condorcet winner, when there is one, but of course it cannot invent one if there is none. Also, there may be ties lower down in the order, with an overall winner and two candidates tied up in second place. For example, there could be an election scenario in which $D$ would beat $B$, $C$, and $D$, while $A$ would beat $C$, $B$ would beat $D$, and $C$ would beat nobody. $D$ would be the winner, and $A$ and $B$ would both get second place.

Now let's return to the example that got us started on the Schulze method. Recall that we had the following ballots for three candidates, $A$, $B$, and $C$:

$$\begin{aligned} 10 \times [A, B, C] \\ 5 \times [B, C, A] \\ 5 \times [C, A, B] \end{aligned}$$

We found that with simple comparisons, without using the Schulze method, $A$ is preferred over $B$ by 15 to 5, $B$ is preferred over $C$ by 15 to 5, and $C$ and $A$ tie up with 10 each. Because both $A$ and $B$ have a pairwise win, we ended up with a tie. How does the Schulze method fare? The preference array for the candidates is:

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 0 & 15 & 10 \\ 5 & 0 & 15 \\ 10 & 5 & 0 \end{bmatrix} \end{array}$$

From that we get the adjacency matrix for the election graph:

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} -\infty & 10 & -\infty \\ -\infty & -\infty & 10 \\ -\infty & -\infty & -\infty \end{bmatrix} \end{array}$$

The graph itself is in figure 10.5. It's easy to see that there are two (strongest) paths starting from $A$, the path $A \to B$ and the path $A \to B \to C$, whereas there is only one strongest path from $B$, $B \to C$. As there are no reverse paths to compare, the Schulze method will give as a result that $A$ wins over both $B$ and $C$, $B$ wins over $C$, and $C$ beats nobody. It will therefore declare $A$ as a winner, resolving the earlier tie. The Schulze method will in general produce fewer ties than a simple pairwise comparison. It can be proven that it meets the criterion of *resolvability*, which means that there is a low possibility of tied results.

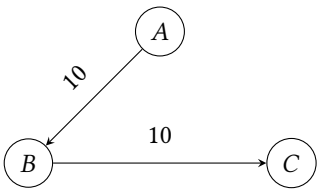



**Figure 10.5**
Another election graph.

Caveat emptor. We have presented the Condorcet criterion and showed that it is a reasonable requirement for an election. We have also presented the Schulze method, which is a reasonable voting method that meets the Condorcet criterion. This does not mean that the Schulze method is the one best voting method, or even that the Condorcet criterion is the one criterion that should guide a polity's decisions as it looks to decide how to run its elections. It can be proven, as it was by the Nobel prize winner Kenneth Joseph Arrow in his PhD thesis, that there exists no perfect voting system where voters express their preferences in the ballots. That is called Arrow's Impossibility Theorem. What is important is that the choice of a voting method should be an informed one, and not made unthinkingly, or perhaps guided by custom. Voters must decide how to vote wisely. That said, the Schulze method does provide a nice example to introduce a graph algorithm, and that is why we chose it here.

## 10.3 The Floyd-Warshall Algorithm

Algorithm 10.2 for calculating the strongest paths is a variant of a venerable algorithm for calculating the shortest paths between all pairs in a graph, the Floyd-Warshall algorithm. It was published by Robert Floyd, but similar algorithms have been published by Bernard Roy and Stephen Warshall. This is algorithm 10.4. Like algorithm 10.2, this also runs in time $\Theta(n^3)$. That makes it in general slower than the Dijkstra algorithm, but it performs well in dense graphs. It is also simple to implement, with no requirements for any special data structures, and it works in graphs with negative weights as well. Attention is needed in line 16 to watch for overflows in computer languages that have no notion of infinity and you use big numbers to stand in for them. In these settings you need to check whether $dist[i,k]$ and $dist[k,j]$ are not equal